9 Pandas 1: Introduction

Lab Objective: *Though NumPy and SciPy are powerful tools for numerical computing, they lack some of the high-level functionality necessary for many data science applications. Python's* pandas *library, built on NumPy, is designed specifically for data management and analysis. In this lab, we introduce pandas data structures, syntax, and explore its capabilities for quickly analyzing and presenting data.*

Note

This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas1.ipynb`. To make the data files accessible, run the following at the top of the lab:

```
>>> from google.colab import files

>>> uploaded = files.upload()
```

This will prompt you upload files for this notebook. For this lab, upload `budget.csv` and `crime_data.csv`.

Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas1.py` file.

Pandas Basics

Pandas is a python library used primarily to analyze data. It combines functionality of NumPy, MatPlotLib, and SQL to create a easy to understand library that allows for the manipulation of data in various ways. In this lab, we focus on the use of Pandas to analyze and manipulate data in ways similar to NumPy and SQL.

Pandas Data Structures

Series

The first pandas data structure is a `Series`. A `Series` is a one-dimensional array that can hold any datatype, similar to a `ndarray`. However, a `Series` has an `index` that gives a label to each entry. An `index` generally is used to label the data.

Typically a `Series` contains information about one feature of the data. For example, the data in a `Series` might show a class's grades on a test and the `Index` would indicate each student in the class. To initialize a `Series`, the first parameter is the data and the second is the index.

```
>>> import pandas as pd
>>>
>>> # Initialize Series of student grades
>>> math = pd.Series(np.random.randint(0,100,4), ['Mark', 'Barbara', 'Eleanor',↩
     'David'])
>>> english = pd.Series(np.random.randint(0,100,5), ['Mark', 'Barbara', 'David'↩
    , 'Greg', 'Lauren'])
```

DataFrame

The second key pandas data structure is a `DataFrame`. A `DataFrame` is a collection of multiple `Series`. It can be thought of as a 2-dimensional array, where each row is a separate datapoint and each column is a feature of the data. The rows are label with an `index` (as in a `Series`) and the columns are labelled in the attribute `columns`.

There are many different ways to initialize a `DataFrame`. One way to initialize a `DataFrame` is passing in a dictionary as the data of the `DataFrame`. The keys of the dictionary will become the labels in `columns` and the values are the `Series` associated with the label.

```
>>> # Create a DataFrame of student grades
>>> grades = pd.DataFrame({"Math": math, "English": english}
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0      NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

Notice that `pd.DataFrame` automatically lines up data from both `Series` that have the same index. If the data only appears in one of the `Series`, the entry for the second `Series` is `NaN`.

We can also initialize a `DataFrame` with a NumPy array. In this way, the data is passed in as a 2-dimensional NumPy array, while the column labels and index are passed in as parameters. The first column label goes with the first column of the array, the second with the second, etc. The same holds for the index.

```
>>> import numpy as np
>>> # Initialize DataFrame with NumPy array
>>> data = np.array([[52.0, 73.0], [10.0, 39.0], [35.0, np.nan], [np.nan, ↩
    26.0], [np.nan, 99.0], [81.0, 68.0]])
>>> grades = pd.DataFrame(data, columns = ['Math', 'English'], index = ['↩
    Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'])
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0     NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

A `DataFrame` can also be viewed as a NumPy array using the attribute `values`.

```
>>> # View the DataFrame as a NumPy array
>>> grades.values
array([[ 52.,  73.],
       [ 10.,  39.],
       [ 35.,  nan],
       [ nan,  26.],
       [ nan,  99.],
       [ 81.,  68.]])
```

Problem 1. Write a function `random_dataframe()` that accepts a dictionary `d` which defaults to `None`. If a dictionary is passed in, initialize a Pandas `DataFrame`. Return a tuple of the attributes `index`, `columns`, and `values` of the `DataFrame`.

If a dictionary is not passed in, generate random data as a `ndarray` and initialize a `DataFrame`. The columns of the `DataFrame` should be the letters `'A'` through `'E'`. The index of the `DataFrame` should be the roman numerals 1-6. Return a tuple of the attributes `index`, `columns`, and `values` of the `DataFrame`.

(Hint: What should the dimension of the data be if no dictionary is passed in?)

Data I/O

The pandas library has functions that make importing and exporting data simple. The functions allow for a variety of file formats to be imported and exported, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
`to_csv()`	Write the index and entries to a CSV file
`to_json()`	Convert the object to a JSON string
`to_pickle()`	Serialize the object and store it in an external file
`to_sql()`	Write the object data to an open SQL database

Table 9.1: Methods for exporting data in a pandas `Series` or `DataFrame`.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a `DataFrame`, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- `delimiter`: The character that separates data fields. It is often a comma or a whitespace character.
- `header`: The row number (0 indexed) in the CSV file that contains the column names.
- `index_col`: The column (0 indexed) in the CSV file that is the index for the `DataFrame`.
- `skiprows`: If an integer n, skip the first n rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- `names`: If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list.

Data Manipulation

Accessing Data

While array slicing can be used to access data in a `DataFrame`, it is always preferable to use the `loc` and `iloc` indexers. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than slicing because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc`/`iloc` explicitly, bypasses the extra checks. The `loc` index selects rows and columns based on their labels, while `iloc` selects them based on their integer position. When using these indexers, the first and second arguments refer to the rows and columns, respectively, just as array slicing.

```
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0     NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0

>>> # Use loc to select the Math scores of David and Greg
>>> grades.loc[['David', 'Greg'],'Math']
David    10.0
Greg      NaN
Name: Math, dtype: float64

>>> # Use iloc to select the Math scores of David and Greg
>>> grades.iloc[[1,3], 0]
David    10.0
Greg      NaN
```

An entire column of a `DataFrame` can be accessed using simple square brackets and the name of the column. In addition, to create a new column or reset the values of an entire column, simply call this column in the same fashion and set the value.

```
>>> # Set new History column with array of random values
>>> grades['History'] = np.random.randint(0,100,6)
>>> grades['History']
Barbara     4
David      92
Eleanor    25
Greg       79
Lauren     82
Mark       27
Name: History, dtype: int64

>>> # Reset the column such that everyone has a 100
>>> grades['History'] = 100.0
>>> grades
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0      NaN    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
Mark     81.0     68.0    100.0
```

Often datasets can be very large and difficult to visualize. Pandas offers various methods to make the data easier to visualize. The methods **`head`** and **`tail`** will show the first or last n data points, respectively, where n defaults to 5. The method **`sample`** will draw n random entry of the dataset, where n defaults to 1.

```
>>> # Use head to see the first n rows
>>> grades.head(n=2)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0

>>> # Use sample to sample a random entry
>>> grades.sample()
        Math  English  History
Lauren   NaN     99.0    100.0
```

It may also be useful to re-order the columns or rows or sort according to a given column.

```
>>> # Re-order columns
>>> grades.reindex(columns['English','Math','History'])
         English  Math  History
```

```
Barbara     73.0  52.0    100.0
David       39.0  10.0    100.0
Eleanor      NaN  35.0    100.0
Greg        26.0   NaN    100.0
Lauren      99.0   NaN    100.0
Mark        68.0  81.0    100.0

>>> # Sort descending according to Math grades
>>> grades.sort_values('Math', ascending=False)
         Math  English  History
Mark     81.0     68.0    100.0
Barbara  52.0     73.0    100.0
Eleanor  35.0      NaN    100.0
David    10.0     39.0    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
```

Other methods used for manipulating `DataFrame` and `Series` panda structures can be found in Table 9.2.

Method	Description
`append()`	Concatenate two or more `Series`.
`drop()`	Remove the entries with the specified label or labels
`drop_duplicates()`	Remove duplicate values
`dropna()`	Drop null entries
`fillna()`	Replace null entries with a specified value or strategy
`reindex()`	Replace the index
`sample()`	Draw a random entry
`shift()`	Shift the index
`unique()`	Return unique values

Table 9.2: Methods for managing or modifying data in a pandas `Series` or `DataFrame`.

Problem 2. The file `budget.csv` contains the budget of a college student over the course of 4 years. Write a function `prob2()` reads in `budget.csv` as a `DataFrame`. Perform the following operations:

1. Reindex the columns such that amount spent on food is the first column and all other columns maintain the same ordering.
2. Sort the `DataFrame` in descending order based on how much money was spent on `Groceries`
3. Reset all values in the `'Rent'` column to `800.0`
4. Reset all values in the first 5 data points to `0.0`

Return the values of the updated `DataFrame` as a NumPy array.

Basic Data Manipulation

Because the primary pandas data structures are subclasses of `ndarray`, most NumPy functions work with pandas structure. For example, basic vector operations work as expected:

```
>>> # Sum history and english grades of all students
>>> grades['English'] + grades['History']
Barbara    173.0
David      139.0
Eleanor      NaN
Greg       126.0
Lauren     199.0
Mark       168.0
dtype: float64

>>> # Double all Math grades
>>>  grades['Math']*2
Barbara    104.0
David       20.0
Eleanor     70.0
Greg         NaN
Lauren       NaN
Mark       162.0
Name: Math, dtype: float64
```

In addition to arithmetic, `Series` have a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 9.3.

Method	Returns
`abs()`	Object with absolute values taken (of numerical data)
`idxmax()`	The index label of the maximum value
`idxmin()`	The index label of the minimum value
`count()`	The number of non-null entries
`cumprod()`	The cumulative product over an axis
`cumsum()`	The cumulative sum over an axis
`max()`	The maximum of the entries
`mean()`	The average of the entries
`median()`	The median of the entries
`min()`	The minimum of the entries
`mode()`	The most common element(s)
`prod()`	The product of the elements
`sum()`	The sum of the elements
`var()`	The variance of the elements

Table 9.3: Numerical methods of the `Series` and `DataFrame` pandas classes.

Basic Statistical Functions

The pandas library allows us to easily calculate basic summary statistics of our data, useful when we want a quick description of the data. The `describe()` function outputs several such summary statistics for each column in a DataFrame:

```
>>> # Use describe to better understand the data
>>> grades.describe()
            Math   English  History
count   4.000000   5.00000      6.0
mean   44.500000  61.00000    100.0
std    29.827281  28.92231      0.0
min    10.000000  26.00000    100.0
25%    28.750000  39.00000    100.0
50%    43.500000  68.00000    100.0
75%    59.250000  73.00000    100.0
max    81.000000  99.00000    100.0
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available.

```
>>> # Find the average grade for each student
>>> grades.mean(axis=1)
Barbara    75.000000
David      49.666667
Eleanor    67.500000
Greg       63.000000
Lauren     99.500000
Mark       83.000000
dtype: float64

>>> # Solve for the unbiased variance between subjects
>>> grades.cov()
               Math  English  History
Math     889.666667    557.0      0.0
English  557.000000    836.5      0.0
History    0.000000      0.0      0.0

>>> # Give correlation matrix between subjects
>>> grades.corr()
            Math  English  History
Math     1.00000  0.84996      NaN
English  0.84996  1.00000      NaN
History      NaN      NaN      NaN
```

The method rank gives a ranking based on methods such as average, minimum, and maximum. This method defaults ranking in ascending order (the least will be ranked 1 and the greatest will be ranked the highest number).

```
>>> # Rank each student's performance based on their highest grade in any class↩
     in descending order
>>> grades.rank(axis=1,method='max',ascending=False)
         Math  English  History
```

```
Barbara   3.0      2.0      1.0
David     3.0      2.0      1.0
Eleanor   2.0      NaN      1.0
Greg      NaN      2.0      1.0
Lauren    NaN      2.0      1.0
Mark      2.0      3.0      1.0
```

These methods can be very effective in interpreting data. For example, the `rank` example above shows use that Barbara does best in History, then English and then Math.

Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. If we are not interested in the missing values, we can simply drop them from the data altogether:

```
>>> # Grades with all NaN values dropped
>>> grades.dropna()
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Mark     81.0     68.0    100.0
```

This is not always the desired behavior, however. Missing data could actually correspond to some default value, such as zero. For example, in the budget dataset, filling `NaN` value with 0 indicates that no money was spent on that item. In the grade dataset, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 50.0
>>> grades.fillna(50.0)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0     50.0    100.0
Greg     50.0     26.0    100.0
Lauren   50.0     99.0    100.0
Mark     81.0     68.0    100.0
```

When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using. For example, `sum()` and `mean()` ignore NaN values in the computation.

Achtung!

Always consider missing data carefully when analyzing a dataset. It may not always be helpful to drop the data or fill it in with a random number. Consider filling the data with the mean of surrounding data or the mean of the feature in question. Overall, the choice for how to fill missing data should make sense with the dataset.

Problem 3. Write a function prob3() that uses budget.csv to answer the questions "Which category affects living expenses the most? Which affects other expenses the most? How much is generally spent in these two categories?". Use the functions above to manipulate the data to perform the following manipulations:

1. Fill all NaN values with 0.0.
2. Create two new columns, 'Living Expenses' and 'Other'. Sum the columns 'Rent' , 'Groceries', 'Gas' and 'Utilities' and set as the value of 'Living Expenses'. Sum the columns 'Dining Out', 'Out With Friends' and 'Netflix' and set as the value of 'Other'.
3. Identify which column correlates most with 'Living Expenses' and which correlates most with 'Other'. This can indicate which columns in the budget affects the overarching categories the most.

Return the mean of each the two columns found in 3 as a tuple. The first mean should be of the column corresponding to 'Living Expenses' and the second to 'Other'.

SQL Operations in pandas

DataFrames are tabular data structures bearing an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases; however, pandas can accomplish many of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, eliminating the need to switch between programming languages for different tasks. Within pandas, we can handle both the querying *and* data analysis.

For the examples below, we will use the following data:

```
>>> name = ['Mylan', 'Regan', 'Justin', 'Jess', 'Jason', 'Remi', 'Matt', '↩
    Alexander', 'JeanMarie']
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ['Sp', 'Se', 'Fr', 'Se', 'Sp', 'J', 'J', 'J', 'Se']
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({'ID': ID, 'Name': name, 'Sex': sex, 'Age': age,↩
     'Class': rank})
>>> otherInfo = pd.DataFrame({'ID': ID, 'GPA': GPA, 'Financial_Aid': aid})
>>> mathInfo = pd.DataFrame({'ID': mathID, 'Grade': mathGd, 'Math_Major': major↩
    })
```

Before querying our data, it is important to know some of its basic properties, such as number of columns, number of rows, and the datatypes of the columns. This can be done by simply calling the `info()` method on the desired `DataFrame`:

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade         5 non-null float64
ID            5 non-null int64
Math_Major    5 non-null object
dtypes: float64(1), int64(1), object(1)
```

SQL SELECT statements can be done by column indexing. WHERE statements can be included by adding masks (just like in a NumPy array). The method `isin()` can also provide a useful WHERE statement. This method accepts a list, dictionary, or `Series` containing possible values of the `DataFrame` or `Series`. When called upon, it returns a `Series` of booleans, indicating whether an entry contained a value in the parameter pass into `isin()`.

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   29

>>> # SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> mask = otherInfo['Financial_Aid'] == 'y'
>>> otherInfomask][['ID', 'GPA']]
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>>  studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
0        Mylan
4        Jason
5         Remi
6         Matt
7    Alexander
```

```
Name: Name, dtype: object
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function. `merge` takes the two `DataFrame` objects to join as parameters, as well as keyword arguments specifying the column on which to join, along with the type (left, right, inner, outer).

```
>>> # SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = ↩
    mathInfo.ID
>>> pd.merge(studentInfo, mathInfo, on='ID') # INNER JOIN is the default
   Age Class  ID    Name Sex  Grade Math_Major
0   20    Sp   0   Mylan   M    4.0          y
1   21    Se   1   Regan   F    3.0          n
2   22    Se   3    Jess   F    4.0          n
3   20     J   5    Remi   F    3.5          y
4   20     J   6    Matt   M    3.0          n
[5 rows x 7 columns]

>>> # SELECT GPA, Grade FROM otherInfo FULL OUTER JOIN mathInfo ON otherInfo.ID↩
     = mathInfo.ID
>>> pd.merge(otherInfo, mathInfo, on='ID', how='outer')[['GPA', 'Grade']]
   GPA  Grade
0  3.8    4.0
1  3.5    3.0
2  3.0    NaN
3  3.9    4.0
4  2.8    NaN
5  2.9    3.5
6  3.8    3.0
7  3.4    NaN
8  3.7    NaN
[9 rows x 2 columns]
```

Problem 4. Read in the file `crime_data.csv` as a pandas object. The file contains data on types of crimes in the U.S. from 1960 to 2016. Set the index as the column 'Year'.

Create a new column `Rate` which contains the crime rate for each year. Using panda commands, find the number of murders in the years where the crime rate was greater than 5% and the number of `Violent` was more than on average. Return an array containing the number of murders in these years.

(Hint: To do `AND` statements, use two masks. Use `values` attribute to create array.)

Problem 5. Answer the following questions using the file `crime_data.csv` and the pandas methods learned in this lab. The answer of each question should be saved as indicated. Return the answers to each question as a tuple (i.e. `(answer_1,answer_2,answer_3)`).

1. Identify the three crimes that have a mean over 1,500,000. Of these three crimes, which two are very correlated? Which of these two crimes has a greater maximum value? Save the title of this column as a variable to return as the answer.
2. Examine the data since 2000. Sort this data (in ascending order) according to number of murders. SELECT Aggravated Assault WHERE Aggravated Assault is greather than 850,000. Save the reordered and SQL queried `DataFrame` as a NumPy array to return as the answer.
3. What decade had the most crime? In this decade, which crime was committed the most? What percentage of the total crime that year was it? Save this value as a float.